8 Масштабируемость компонентов React

Посмотрите вступительный видеоролик к этой главе, отсканировав QR-код или перейдя по ссылке http://reactquickly.co/videos/ch08.

ЭТА ГЛАВА ОХВАТЫВАЕТ СЛЕДУЮЩИЕ ТЕМЫ:

- Свойства по умолчанию в компонентах.
- Типы свойств React и проверка данных.
- Рендеринг дочерних элементов.
- Создание компонентов более высокого порядка React для повторного использования кода.
- Лучшие практики: презентационные и контейнерные компоненты.

К настоящему моменту вы узнали, как создавать компоненты, как сделать их интерактивными и как работать с пользовательским вводом (событиями и элементами ввода). С этими знаниями вы прошли значительную часть пути к построению сайтов с компонентами React, и все же время от времени неизбежно будут возникать раздражающие проблемы. Это особенно часто происходит в больших проектах, в которых вы полагаетесь на компоненты, созданные другими разработчиками (вашими коллегами или соавторами в проектах с открытым кодом).

Например, если вы используете компонент, написанный кем-то другим, как узнать, предоставляете ли вы правильные свойства для него? И как использовать существующий компонент с небольшой дополнительной функциональностью (которая также применяется к другим компонентам)? Все эти проблемы относятся к области *масштабируемости разработки*, то есть к работе с кодом в процессе разрастания кодовой базы. Некоторые возможности и паттерны React помогут вам в этом.

Эти темы важны, если вы хотите узнать, как эффективно построить сложное приложение React. Например, компоненты более высокого порядка позволяют расширить функциональность компонента, а типы свойств предоставляют безопасность проверки типов и, в определенной степени, — защиту на уровне здравого смысла.

К концу этой главы вы будете знать большинство возможностей React из этой области. Вы узнаете, как сделать ваш код более удобным для разработчика (при помощи типов свойств) и как повысить эффективность вашей работы (при помощи имен компонентов и компонентов более высокого порядка). Возможно, коллеги будут восхищаться элегантностью ваших решений. Все эти возможности помогают более эффективно использовать React, так что перейдем без лишних слов к делу.

ПРИМЕЧАНИЕ Исходный код примеров этой главы доступен по адресам www.manning.com/books/react-quickly и https://github.com/azat-co/react-quickly/tree/master/ch08 (в папке ch08 репозитория GitHub https://github.com/azat-co/react-quickly). Также некоторые демонстрационные примеры доступны по адресу http://reactquickly.co/demos.

8.1. Свойства по умолчанию в компонентах

Представьте, что вы строите компонент `Datepicker`, который получает несколько обязательных свойств, например количество строк, локальный контекст и текущую дату:

```
<Datepicker currentDate={Date()} locale="US" rows={4}/>
```

Что произойдет, если новый участник команды попытается воспользоваться вашим компонентом, но забудет передать необходимое свойство `currentDate`? Или если другой коллега передаст строку `"4"` вместо числа 4? Компонент не сделает ничего (значения не определены) или произойдет нечто похуже — он может вызвать фатальный сбой, и во всем обвинят вас (`ReferenceError`?).

Как ни печально, такая ситуация не так уж редко встречается в веб-разработке, потому что JavaScript относится к числу языков со слабой типизацией. К счастью, React предоставляет возможность определения значений по умолчанию для свойств — статический атрибут `defaultProps`.

Главное преимущество `defaultProps` заключается в том, что при отсутствии свойства будет сгенерировано значение по умолчанию. Чтобы задать значение свойства по умолчанию для класса компонента, определите `defaultProps`. Например, в упомянутом выше определении компонента `Datepicker` можно добавить статический атрибут класса (не атрибут экземпляра, который не сработает, — атрибуты экземпляров задаются в `constructor()`):

```
class Datepicker extends React.Component {
  ...
}
```

```
Datepicker.defaultProps = {
  currentDate: Date(),
  rows: 4,
  locale: 'US'
}
```

Чтобы продемонстрировать применение `defaultProps`, представьте, что у вас имеется компонент, который рендерит кнопку. Обычно кнопки снабжаются надписями, но эти надписи должны настраиваться. Если специальное значение не указано, на кнопке должно выводиться значение по умолчанию.

Надпись кнопки определяется свойством `buttonLabel`, которое используется в атрибуте `return` метода `render()`. Это свойство всегда должно содержать `Submit`, если значение не задано. Для этого вы реализуете статический атрибут класса `defaultProps`, который представляет собой объект со свойством `buttonLabel` и значением по умолчанию:

```
class Button extends React.Component {
  render() {
    return <button className="btn" >{this.props.buttonLabel}</button>
  }
}
Button.defaultProps = {buttonLabel: 'Submit'}
```

Родительский компонент `Content` рендерит четыре кнопки. У трех из этих четырех кнопок отсутствуют свойства:

```
class Content extends React.Component {
  render() {
    return (
      <div>
        <Button buttonLabel="Start"/>
        <Button />
        <Button />
        <Button />
      </div>
    )
  }
}
```

Сможете угадать результат? На первой кнопке выводится надпись Start, а на остальных кнопках — надпись Submit (рис. 8.1).

Определять значения свойств по умолчанию стоит почти всегда, потому что они сделают ваши компоненты более устойчивыми к ошибкам. Другими словами, ваши компоненты становятся умнее, потому что они будут обладать минимальным оформлением и поведением, даже если необходимые данные не были переданы.

Также на происходящее можно взглянуть иначе: определение значения по умолчанию означает, что вам не придется объявлять старое значение снова и снова. Если

вы используете одно значение свойства в большинстве случаев, но при этом хотите оставить возможность изменить это значение (переопределить значение по умолчанию), используйте `defaultProps`. Переопределение значения по умолчанию не создает никаких проблем, как наглядно показывает первая кнопка в этом примере.

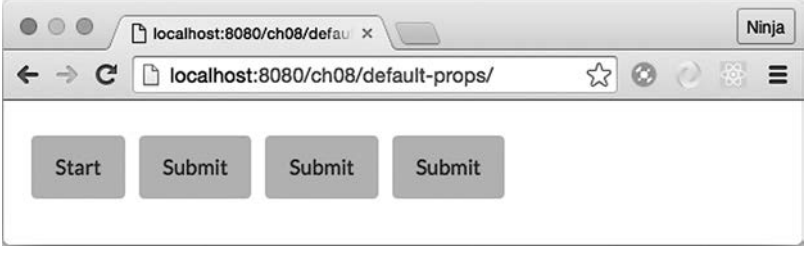

Рис. 8.1. Надпись на первой кнопке была задана при ее создании. У других элементов ее нет, поэтому они используют значение свойства по умолчанию

8.2. Типы свойств React и проверка данных

Возвращаясь к приведенному ранее примеру с компонентом `Datepicker` и коллегами, не знавшими о типах свойств (`"4"` вместо 4), вы можете задать типы свойств, которые должны использоваться с классами компонентов React.js. Для этой цели используется статический атрибут `propTypes`. Он не обеспечивает форсированной проверки типов значений свойств, а всего лишь выдает предупреждение. Таким образом, если вы находитесь в режиме разработки, а тип не совпадает, вы получите предупреждающее сообщение в консоли, и в процессе эксплуатации приложения ничего не будет сделано для того, чтобы помешать использованию неправильного типа. Фактически React.js подавляет это предупреждение в режиме эксплуатации. Таким образом, `propTypes` — в основном вспомогательная функция, которая предупредит вас о несоответствии типов данных на стадии разработки.

REACT В РЕЖИМЕ РАЗРАБОТКИ И ЭКСПЛУАТАЦИИ

Команда React.js определяет режим разработки как режим, использующий не минифицированную версию React (то есть без сжатия), а режим эксплуатации — как режим, использующий минифицированную версию. От авторов React:

«Мы предоставляем две версии React: версию без сжатия для разработки и минифицированную версию для реальной эксплуатации. Версия для разработки включает дополнительные предупреждения о распространенных ошибках, а версия для эксплуатации включает дополнительные оптимизации быстродействия и подавляет все сообщения об ошибках».

Для React 15.5 и более поздних версий (в большинстве примеров из книги используется React v15.5) определения типов предоставляются отдельным пакетом `prop-types` (www.npmjs.com/package/prop-types). Вы должны включить `prop-types` в свой файл HTML. Пакет становится глобальным объектом (`window.PropTypes`):

```
<!-- версия для разработки -->
<script src="https://unpkg.com/prop-types/prop-types.js"></script>
<!-- версия для эксплуатации -->
<script src="https://unpkg.com/prop-types/prop-types.min.js"></script>
```

Если вы используете React 15.4 или более раннюю версию, включать `prop-types` не нужно, потому что типы встроены в React: `React.propTypes`.

Рассмотрим простой пример определения статического атрибута `propTypes` для класса `Datepicker` с разными типами: строковым, числовым и перечисляемым. В этом примере используется React v15.5, а `prop-types` включается в разметку HTML (здесь не показано):

```
class Datepicker extends React.Component {
  ...
}
Datepicker.propTypes = {
  currentDate: PropTypes.string,
  rows: PropTypes.number,
  locale: PropTypes.oneOf(['US', 'CA', 'MX', 'EU'])
}
```

window.PropTypes, потому что сценарий включает prop-types.js

> **ПРЕДУПРЕЖДЕНИЕ** Никогда не полагайтесь на проверку вводимых данных в клиентской части, потому что ее легко можно обойти. Используйте ее только для улучшения опыта взаимодействия и выполняйте все проверки на стороне сервера.

Для проверки типов свойств используйте свойство `propTypes` с объектом, содержащим свойства как ключи, а типы как значения. Типы React.js содержатся в объекте `PropTypes`:

- `PropTypes.string`
- `PropTypes.string`
- `PropTypes.number`
- `PropTypes.bool`
- `PropTypes.object`
- `PropTypes.array`
- `PropTypes.func`
- `PropTypes.shape`
- `PropTypes.any.isRequired`

- `PropTypes.objectOf(PropTypes.number)`
- `PropTypes.arrayOf(PropTypes.number)`
- `PropTypes.node`
- `PropTypes.instanceOf(Message)`
- `PropTypes.element`
- `PropTypes.oneOfType([PropTypes.number, ...])`

Для демонстрации расширим пример `defaultProps`, добавив несколько типов свойств в дополнение к значениям свойств по умолчанию. Этот проект имеет аналогичную структуру: content.jsx, button.jsx и script.jsx. Файл index.html содержит ссылку на prop-types.js:

```
<!DOCTYPE html>
<html>

  <head>
    <script src="js/react.js"></script>
    <script src="js/prop-types.js"></script>
    <script src="js/react-dom.js"></script>
    <link href="css/bootstrap.css" type="text/css" rel="stylesheet"/>
    <link href="css/style.css" type="text/css" rel="stylesheet"/>

  </head>

  <body>
    <div id="content" class="container"></div>
    <script src="js/button.js"></script>
    <script src="js/content.js"></script>
    <script src="js/script.js"></script>
  </body>

</html>
```

Определим класс `Button` с необязательным текстом строкового типа. Чтобы реализовать его, следует определить статический атрибут класса (свойство этого класса) `propTypes` с ключом `title` и значением `PropTypes.string`. Этот код сохраняется в файле button.js:

```
Button.propTypes = {
  title: PropTypes.string
}
```

Также свойство можно объявить обязательным. Для этого добавьте к типу `isRequired`. Например, свойство `title` является обязательным и относится к строковому типу:

```
Button.propTypes = {
  title: PropTypes.string.isRequired
}
```

Кнопка также требует определения свойства `handler`, значением которого является функция (в конце концов, от кнопок без действия особой пользы не было):

```
Button.propTypes = {
  handler: PropTypes.func.isRequired
}
```

Здесь удобно то, что вы можете определить собственную процедуру проверки данных. Чтобы реализовать нестандартную проверку, достаточно создать выражение, которое возвращает экземпляр `Error`, а затем использовать это выражение в `propTypes: {..}` как значение свойства. Например, следующий код проверяет свойство `email` с использованием регулярного выражения из `emailRegularExpression` (которое я скопировал из интернета — а значит, оно должно быть правильным, верно[1]?):

```
...
propTypes = {
    email: function(props, propName, componentName) {
      var emailRegularExpression =
    /^([\w-]+(?:\.[\w-]+)*)@((?:[\w-]+\.)*\w[\w-]{0,66})\.([a-z]{2,6}(?:\.
     ➥ [a-z]{2})?)$/i
      if (!emailRegularExpression.test(props[propName])) {
        return new Error('Email validation failed!')
      }
    }
}
...
```

А теперь соберем все воедино. Компонент `Button` будет вызываться со свойством `title` (строка) и без него, а также со свойством `handler` (необходимая функция). В листинге 8.1 (ch08/prop-types) типы свойств помогают проверить, что `hander` является функцией, `title` — строкой, а `email` соответствует переданному регулярному выражению.

Листинг 8.1. Использование propTypes и defaultProps

```
class Button extends React.Component {
  render() {
    return <button className="btn">{this.props.buttonLabel}</button>
  }
}

Button.defaultProps = {buttonLabel: 'Submit'}

Button.propTypes = {
  handler: PropTypes.func.isRequired,
```

Требует свойства handler со значением функции

[1] Существует много версий регулярного выражения для адресов электронной почты в зависимости от строгости проверки, доменных зон и других критериев. См. «Email Address Regular Expression That 99,99 % Works», http://emailregex.com; «Validate email address in JavaScript?» (вопрос на сайте Stack Overflow), http://mng.bz/zm37; и Regular Expression Library, http://regexlib.com/Search.aspx?k=email.

```
  title: PropTypes.string,
  email(props, propName, componentName) {
    let emailRegularExpression =
    /^([\w-]+(?:\.[\w-]+)*)@((?:[\w-]+\.)*\w[\w-]{0,66})\.([a-z]{2,6}(?:\.
    ➥ [a-z]{2})?)$/i
    if (!emailRegularExpression.test(props[propName])) {
      return new Error('Email validation failed!')
    }
  }
}
```

Определяет проверку адреса электронной почты по регулярному выражению

Определяет необязательное свойство title со строковым значением

Теперь реализуем родительский компонент `Content`, который рендерит шесть кнопок для тестирования предупреждений, генерируемых для типов свойств (ch08/prop-types/jsx/content.jsx).

Листинг 8.2. Рендеринг шести кнопок

```
class Content extends React.Component {
  render() {
    let number = 1
    return (
      <div>
        <Button buttonLabel="Start"/>
        <Button />
        <Button title={number}/>
        <Button />
        <Button email="not-a-valid-email"/>
        <Button email="hi@azat.co"/>
      </div>
    )
  }
}
```

Выдает предупреждение о том, что обработчик отсутствует

Выдает предупреждение о том, что свойство title должно быть строкой

Выдает предупреждение о неправильном формате электронной почты

При выполнении этого кода в консоли выводятся три предупреждения (не забудьте открыть консоль): мои результаты показаны ниже и на рис. 8.2. Первое предупреждение относится к функции `handler`, которая обязательно должна быть задана и которая была опущена для нескольких кнопок:

```
Warning: Failed propType: Required prop `handler` was not specified in
  `Button`. Check the render method of `Content`.
```

Во втором предупреждении говорится о неверном формате электронной почты для четвертой кнопки:

```
Warning: Failed propType: Email validation failed! Check the render method
of `Content`.
```

Третье предупреждение относится к неверному типу значения `title`, которое должно быть строкой (я передал число для одной кнопки):

```
Warning: Failed propType: Invalid prop `title` of type `number` supplied to
`Button`, expected `string`. Check the render method of `Content`.
```

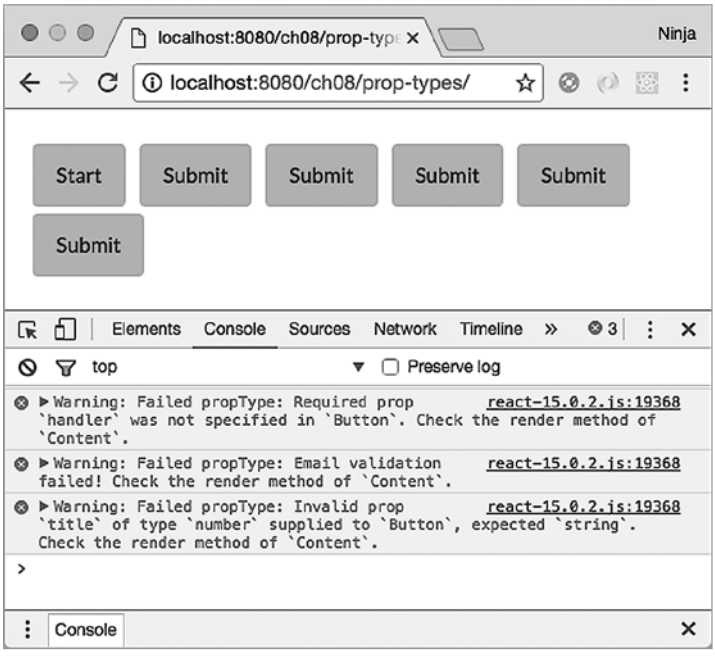

Рис. 8.2. Предупреждения, обусловленные неверным типом свойств

Интересно, что `handler` отсутствует у нескольких кнопок, но вы видите только одно предупреждение. React предупреждает о каждом свойстве только один раз на каждый вызов `render()` для `Content`.

Здесь мне нравится то, что React сообщает, какой родительский компонент следует проверить (`Content` в данном примере). Представьте, что у вас сотни компонентов. Очень полезная информация!

Развернув сообщение в DevTools, вы увидите номер строки элемента `Button`, в которой возникли проблемы и которая привела к выдаче предупреждения. На рис. 8.3 я сначала раскрыл сообщение, а затем нашел свой файл (content.js). В сообщении сказано, что проблема возникла в строке 9.

Если теперь щелкнуть на фрагменте `content.js:9` на консоли, эта строка открывается на вкладке Source (рис. 8.4). Она четко показывает, кто виноват:

```
React.createElement(Button, { title: number }),
```

Вам не понадобятся карты исходного кода (хотя в части 2 вы научитесь создавать и использовать их), чтобы понять, что проблема возникла из-за третьей кнопки.

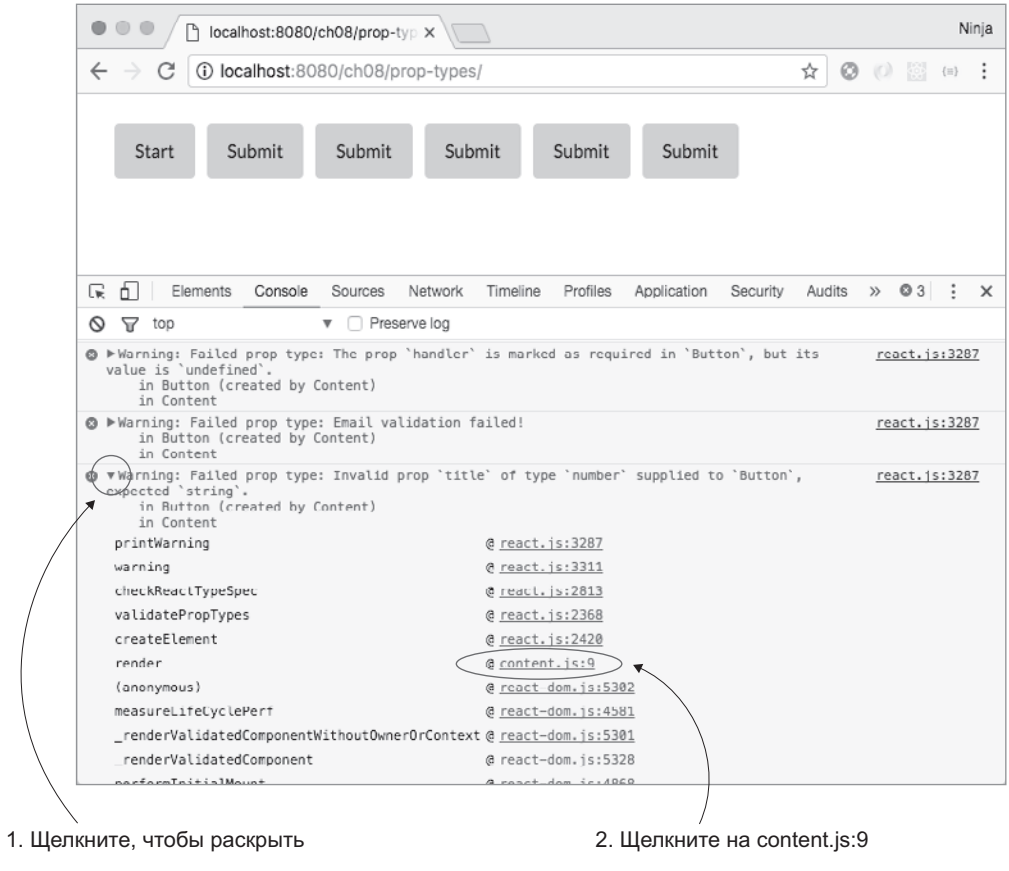

Рис. 8.3. При раскрытии предупреждения виден номер проблемной строки: 9

ПРИМЕЧАНИЕ Еще раз повторю: эти предупреждения выводятся только в неминифицированной версии React (то есть в режиме разработки).

Попробуйте поэкспериментировать с типами свойств и проверкой данных — это удобный механизм. Учтите, что в этом коде используется тот же компонент `Button`:

```
<Button title={number}/>
```

Сможете ли вы обнаружить проблему? Как вы думаете, сколько предупреждений вы получите? (Подсказка: свойства `handler` и `title`.)